INITIALISATION

Use {} to initialize. An error will be thrown if an invalid type is used. Whereas () would try to use coercion:

int n = 6.23; // coerced to int 6

int n {6.23}; // error!

RAII

Resource Allocation Is Initialisation (RAII) is the technique of acquiring resources in a constructor and releasing them in a destructor. Here, resource isn’t just memory; it could be a file, database connection, etc.

Example (not using RAII)

myFile = fopen(“filename.txt”);

// code here

// oops an exception is thrown here

myFile.close(); // release resource but never called because exception thrown above!

Example (using RAII)

Class FileHandler {

Private:

FILE\* file;

Public:

FileHandler(string filename) {

file = fopen(filename);

}

~FileHandler() {

fclose(file);

}

}

// the destructor will be called and the resource released even if an exception is thrown.

FUNCTIONS

Functions returning a pointer to an object allocated on the heap are dangerous. E.g.

Shape\* read\_shape();

Use smart pointers instead:

Unique\_ptr<Shape> read\_shape();

This will delete the object when not needed.

Function return types can also be written like (although not tradition):

auto name(arg) -> int {}

constexpr functions

Evaluate the initializer at compile time, e.g.

Constexpr int fac(int n) { return n; }

Int f5 = fac(5); // may be evaluated at compile time

Int fn = f(n); // error. Evaluated at runtime (n is a variable)

To be evaluated at compile time, a function must be suitably simple; single return type, no loops, no local variables and no side effects.

Inline functions

An inline function must be defined identically in every translation unit in which it is used. Inline functions should be defined in header files and included.

File1.h

Inline int next(int i) { return i + 1;}

File1.cpp

#include “file1.h”

Int h(int i) { return next(i);

STRUCTS  
Remember, structs are classes except members are public by default. Use structs in headers in cpp files exactly as you would classes.

Date.h

Struct Date {

Void init(int dd, int mm, int yy);

}

Date.cpp

Void Date::init(int dd, int mm, int yy) {

// code here

}

LAMBDA

Int sum = 10;

auto x = [sum](int d) mutable -> int {return sum += d; }

mutable, which is optional, means it changes a value.

If a lambda takes no arguments, the argument list can be omitted. [](){} becomes []{}

this keyword

Add *this* in the capture list to access class members.

Auto x = [this]() { // access class members }

CLASSES

When a class is responsible for an object accessed through a pointer, use a copy constructor and move assignment. A move constructor does not take a const argument.

After a move, a moved-from object should be in a state that allows a destructor to be run.

Using the default move or copy for a class in a hierarchy is typically a disaster. We simply don’t know what members the derived class has. The best thing to do is usually to delete the default copy and move.

Copy

ClassName(const ClassName&) = delete;

ClassName& operator=(const ClassName&) = delete;

If you need to copy an object in a class hierarchy, write some kind of clone function.

By default, objects can be copied. The copy of a class object is a copy of each member. If that is not the behaviour wanted, a more appropriate behaviour can be provided.

Implicit vs explicit conversions

Class A {

Public:

A(int n);

}

Class B {

Public:

explicit B(int n);

}

Void hi(A a);

Void bye(B b);

hi(5); // OK, implicit conversion. A instance initialized with ‘5’ as constructor value

bye(5); // error. Explicit conversion

bye(B{2}); // OK.

It is good practice to mark constructors with 1 parameter as explicit.

Copy and move

6 situations in which an object is copied or moved:

Class X {

Public:

X(someType); // ordinary

X() // default

X(const X&); // copy

X(X&&); // move

X& operator= (const X&); // copy assignment: clean up target + copy

X& operator=(X&&); // move assignment: clean up target + move

~X(); // destructor

}

Copy constructors are implicit, e.g.

Class X {

Public:

X(int num);

}

X a; // error. no arg passed

X b{}; // error. no arg passed

X c{1}; // ok

X d{c}; // default copy constructor, ok. As *c* was correctly created, we can copy it implicitly even though no copy constructor explicitly defined.

A copy constructor and a copy assignment differ in that a copy constructor initializes uninitialized memory, whereas the copy assignment must handle an object that has already been constructed and may own resources.

In general, if a class has a pointer member, the default copy and move operations should be considered suspicious.

Destructors

A destructor can be declared as virtual, and usually should be for a class with a virtual function. Failure could cause resources to be leaked.

static member initialization

Static members are declared in classes but have to be defined outside.

Class Node {

// …

Static int node\_count; // declared

}

Int Node::node\_count = 0; // defined

There are a few exceptions to this rule; if the static member is a const or an enum, e.g.

Class Node {

// …

Static const int c1 = 7;

}

Forward declaration

Forward declare when you have cyclic dependencies, e.g. class A depends on class B and vice versa:

*A.cpp*

class B; // forward declare

class A { // implementation }

*B.cpp*

#include “A.h”

Class B { // implementation }

ARRAYS

Arrays cannot directly be passed by value. An array is passed as a pointer to its first element (pointer value):

Void comp(double arg[10])

Same as:

Void comp(double\* arg)

* Avoid multidimensional arrays, define suitable containers instead.
* Avoid passing arrays as pointers, as the size is not available. Passing the size as another argument to a function is a workaround. Instead, pass a container by reference.

Prefer <array> over the built-in array (T[N]):

Array<type, n>; // n must be specified and known at compile time!

// use a <vector> if you need the element count to be variable

POINTERS

\* = contents of

Use nullptr rather than 0 or NULL.

Applying *delete* to nullptr has no effect, so no need to check existence.

Void pointer

A pointer which has no associated data type with it.

Int n = 10;

Void\* ptr = &n;

Cout << static\_cast<int>(\*ptr);

REFERENCES

& = address of

There are 3 kinds of references:

1. Lvalue: to refer to objects whose value we want to change
2. Const (lvalue): to refer to objects we don’t want to change
3. Rvalue: to refer to objects whose values we do not need to preserve after we have used it

Rvalue reference swap:

*Old style*

Void swap(T& a, T& b) {

T tmp {a} // now 2 copies of a

a = b; // now 2 copies of b

b = tmp; // now 2 copies of tmp

}

If T is a type for which it can be expensive to copy elements such as string or vector, this swap function becomes an expensive operation.

*Instead*

Void swap(T& a, T& b) {

T tmp {static\_cast<T&&>(a)};

a = static\_cast<T&&>(b);

b = static\_cast<T&&>(tmp);

}

With the std library you can use std::move() which does this for you, e.g.

T tmp{std::move(a)};

a = std::move(b);

Only use rvalue references for forwarding and move semantics.

POINTERS VS REFERENCES

Use a pointer when:

* You need to change which object to refer to

Use a reference when:

* You want to be sure that a name always refers to the same object
* You want to use a user-defined (overloaded) operator on something that refers to an object

Pass by pointer vs by reference:

* A pointer can be reassigned, a reference cannot
* A pointer can be assigned null, reference cannot
* Pointers can iterate over an array
* A pointer uses -> to access members, references use .
* A pointer needs to be dereferenced with \* to access the memory location it points to, whereas a reference can be used directly.

ENUMERATIONS

Prefer enum classes:

Enum class Color {red, blue, green}

Enum class Traffic {green, yellow, red}

Color col = Color::red;

This makes the enum strongly typed and scoped. We cannot mix up traffic and color ‘red’.

Enumerations are user defined types so we can define operators for them.

enum Day = { sun, mon, tue, wed, thu, fri, sat };

Day& operator++(Day& d) {

return d = (sat == d) ? sun : static\_cast<Day>(d + 1);

}

NAMESPACES

Resolution operator

::Testing

Means look in the global namespace. Similar to PHPs \Exception (\).

EXCEPTIONS

Common std exceptions:

* out\_of\_range
* length\_error
* bad\_alloc

Catching the *std::exception* does not catch all exceptions. Don’t assume that every exception is derived from *std::*exception. E.g. if someone unwisely throws an int it would not caught. Ellipsis will catch all:

catch(…) {}

CONTAINERS

Prefer *auto* when working with iterators:

auto iBegin = vector.begin();

auto iEnd = vector.end();

std::algorithms

std::vector<int> v1 = {3, 2, 1, 5, 4};

std::sort(v1.begin(), b1.end());

std::binary\_search(v1.begin(), v1.end(), 4); // output 0 or 1

std::algorithms using predicates

template<typename T>

bool isEven(T n) { return n % 2 == 0; }

std::vector<int> v1 = {2, 4, 6, 20, 10};

std::all\_off(v1.begin(), v1.end(), isEven<int>); // true. Similar to js *every*

// also have any\_off, none\_off

auto it = find(v1.begin(), v1.end(), 20); // returns iterator. End if not found

if (it != v1.end()) {

cout << “Found at index: “ << it – v1.begin();

}

// also have find\_if (takes predicate as 3rd argument), find\_if\_not

auto c = count(v1.begin(), v1.end(), 2); // count occurrences of 2

// also have count\_if (takes predicate as 3rd argument)

replace(v1.begin(), v1.end(), 5, 99); // replace 5 with 99

// also have replace\_if (takes predicate as 3rd argument)

auto it = remove(v1.begin(), v1.end(), 42); // removes 42.

// returns iterator which points to the last element that was not removed (or the end if nothing removed).

// if element was removed, remember to resize the container

V1.resize(it – v1.begin());

Merge(v1.begin(), v1.end(), v2.begin(), v2.end(), v3.begin());

ERROR HANDLING

Some standard library facilities report errors by throwing exceptions

|  |  |
| --- | --- |
| Bitset | Throws *invalid\_argument, out\_of\_range, overflow\_error* |
| Iostream | Throws *ios\_base::failure* |
| Regex | Throws *regex\_error* |
| String | Throws *length\_error, out\_of\_range* |
| Vector | Throws *length\_error, out\_of\_range* |
| Any container with at() | Throws *out\_of\_range* |
| new T | Throws *bad\_alloc* |
| Dynamic\_cast<T>(r) | Throws *bad\_cast* |
| Typeid() | Throws *bad\_typeid* |
| Thread | Throws *system\_error* |
| Call\_once() | Throws *system\_error* |
| Mutex | Throws system\_error |
| Unique\_lock | Throws *system\_error* |
| Condition\_variable | Throws *system\_error* |
| Async() | Throws system\_error |
| Packaged\_task | Throws bad\_alloc, future\_error |
| Future and promise | Throws *future\_error* |

Runtime error

* range\_error
* overflow\_error
* underflow\_error
* system\_error

Logic error

* length\_error
* domain\_error
* out\_of\_range
* invalid\_argument
* future\_error

COMPILATION / LINKING

Remember, translation units are converted to obj files before linking first, so if you have *int x = 2;* and *int x = 1* in another, a linking error will occur as it is defined twice.

A name that can be used in translation units different from the one in which it was defined is said to have *external linkage*. A name that can be referred to only in the translation unit in which it was defined is said to have *internal linkage*.

static int x = 1; // remove *static* to make external

const char x2 = ‘a’; // precede with *extern* to make external

A variable defined without an initializer in the global or namespace scope is initialized by default. *Int x;* is the same as *int x = 0*;

The linker does not see local variables. They are said to have *no linkage.*

GENERAL

Assertions

assert() // run time

static\_assert() // compile time

static\_assert(e,s); // evaluate *e* at compile time, give *s* as a compiler error message if *!e*

If the NDEBUG macro is defined before the inclusion of <assert.h> then asserts are ignored. This is good for production code.

What can be checked at compile time is generally best checked at compile time using *static\_assert*.

Using std::cin to populate array

Std::vector<int> read(std::istream& is) {

Std::vector<int> v;

For (double d : v) {

v.push\_back(d);

}

Return v;

}

Read(std::cin);

Std::initializer\_list

Used to define the initializer list constructor

Class List {

Public:

List(std::initializer\_list<int> ls) {

// loop through and push to vector perhaps?

}

}

List list = {10, 20, 30, 40, 50, 60}

Using IO for user defined types

Struct Entry {

String name;

Int number;

}

Ostream& operator<<(ostream& os, const Entry& e) {

Return os << e.name << ‘ ‘ << e.number;

}

Cout << entryObject;

Concurrency

Concurrency refers to the execution of several tasks simultaneously.

Tasks are a computation that can potentially be executed concurrently with other computations.

Threads are a system-level representation of a task in a program.

#include <thread>

Std::thread t1 {f, 10} // where f is a function and 10 is f argument

// executes in a separate thread

t1.join() // wait for t1 to finish before exiting program

Threads of a program share a single address space, process do not, this means threads can communicate with each other through shared objects.

Operations for the current thread are found in namespace *this\_thread*:

this\_thread::sleep\_for(t);

this\_thread::get\_id();

// etc.

Time

Time utilities are in the <chrono> header and std::chrono namespace.

auto t0 = high\_resolution\_clock::now();

do\_work();

auto t1 = high\_resolution\_clock::now();

cout << duration\_cast<milliseconds>(t1 – t0).count(); // cast to ms from ns

Type predicates

<type\_traits> header

Int num = 10;

is\_arithmetic<num>(); // true

others:

* is\_class()
* is\_pod()
* is\_literal\_type()
* is\_base\_of()
* etc…

Type aliases

using pchar = char\* // pointer to a char

Use an alias to define a meaningful name for a built-in type in cases which the built-in type used to represent a value might change.

Use enums and classes to define new types.

Const

When using a pointer, 2 objects are involved: the pointer itself, and the object pointed to.

*Easier to understand when read right to left*

Char\* const cp; // const pointer to char

Char const\* pc; // pointer to const char

Const char\* pc2; // pointer to const char

Functor

A functor is a class with an operator() overload

Template<typename T>

Class MultiplyBy {

Private:

T multipier mult;

Public:

MultiplyBy(T n) : mult(n) {}

T operator() (T n) { return mult \* n; }

}

MultiplyBy<int> x(9);

x(5); // 45

x(10); // 90

invariant

You should throw errors in your constructors if invariants cannot be established, e.g.

*The invariant here is that ‘age must be a positive number’:*

Class Person {

Private:

int \_age;

public:

Person(int num) {

If (num < 0) {

Throw InvalidNumber{num};

}

\_age = num;

}

}

If you throw in a constructor, the destructor is not called, so make sure no resources are leaked (e.g. file handles, memory, database handles, etc.).

Dynamic\_cast

*dynamic\_cast<Type\*>(ptr);*

A runtime cast… if ptr can be casted to Type\*, then it will be (e.g. if it is a derived class), else it will return a null ptr.

auto derived = dynamic\_cast<Parent\*>(obj);

if (derived) {

// casted to Parent at runtime

} else {

// unable to cast

}

=> static\_cast is the same but done at compile time.

Volatile

Used to indicate that an object can be modified by something external to the thread of control. Using this tells the compiler not to try and optimize that object.

Generic

* Prefer scoped variables over free-store when possible
* Use pass by value for small objects
* Use pass by const reference to pass large values you don’t want to modify
* Use rvalue references to implement move and forwarding
* Use pass by reference only if you have to
* Use *noexcept* when functions don’t throw exceptions. Be careful using this too much!
* Have *main()* catch and report all exceptions
* Don’t put a *using-directive* (namespace) in a header file
* Put private members last in a class to emphasis the functions providing the public user interface
* A class with a virtual function should have a virtual destructor
* Don’t call virtual functions during construction or destruction. At the point of construction, the object isn’t yet created. Calling a virtual function from a destructor will reflect only what is still not destroyed.
* Type safety means that the compiler can check if you’re using the right type. E.g. *printf(“Hello %s”, 10);* is not type safe as this will compile ok (even though expecting a string). This could result in a runtime error.